PREPRINT UCRL- 84819

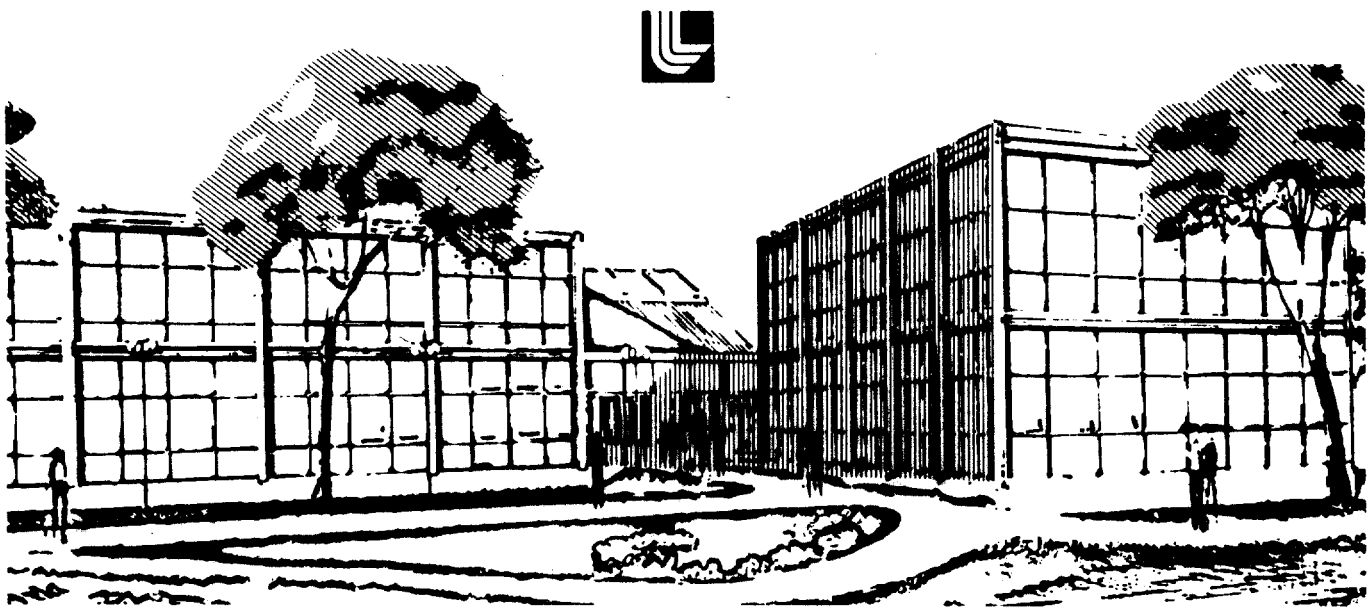
Lawrence Livermore Laboratory

Resource Access Control in a Network Operating System

James E. Donnelley
John G. Fletcher

April 25, 1980

ACM Pacific 1980, San Francisco, CA, November 12-14, 1980



Resource Access Control in a Network Operating System

by J.E. Donnelley
and
J.G. Fletcher

Lawrence Livermore Laboratory

Computer systems being incorporated into mature support networks are facing a substantial protocol implementation effort in granting controlled access to their resources and in obtaining access to network-supplied resources. This protocol implementation effort can be signifigantly reduced by using resource sharing protocols that are independent of specific resource semantics.

A capability-passing model for distributed access control is described and several capability management protocols are discussed. Highlights of the discussion include:

- the inalienable right to pass capabilities,
- capability theft through data theft and reflection,
- capability management by public key encryption,
- a capability passing structure, and
- resource sharing by humans using an integration of network directories.

Keywords and phrases: distributed, network operating system, communication protocol, resource sharing, capability, public key, encryption, access control, directory.

1. Introduction

The fundamental task of a computer operating system (OS) is resource management and control. In the case of stand-alone single-computer operating systems the assumption is made that its resources are created from directly attached peripheral devices (figure 1). In this type of system the operating system is lord and master of its realm and is able to dole out resource access in any way it chooses. The absolute power wielded by these stand-alone systems has given rise to numerous forms of resource access control. These include: user indexes and groups, directories, passwords, access control lists, capability lists, and others [Bob72, RiT74, Org72, Wul74].

In recent years the distributed nature of many modern computing facilities has created a need for distributed operating systems adequate to manage access control for their distributed resources. The component operating system for a single machine in a distributed computing facility (figure 2) generally finds itself more of a demigod than did its omnipotent stand-alone ancestors. These component systems do control their directly attached peripherals but often must share access to most resources with other computers on a communication network (figure 3). The resource sharing mechanisms possible for such component systems are considerably more restricted than those for stand-alone systems.

An example of this situation for component operating systems was that faced when a Cray-1 processor was recently added to the Octopus network [Fle73, Wat78] at the Lawrence Livermore Laboratory. When this processor was attached to the network, its only peripherals were enough disks for temporary file storage and a high speed network interface (figure 2)[Chr77, DoY76]. The task of the component OS for such a machine is to make its processing and disk storage resources available to the network and to allow its processes convenient access to resources available on the network. The design for such an OS becomes in large part the design of a communication protocol: a protocol for distributed access control. We will explore some of the issues and methods involved in the design of such distributed access protocols.

2. Communication Primitives

In order to define an access control protocol it is necessary to specify the communication primitives that it will be built upon. It is important to require as little as possible of these primitives in order to allow the defined protocol to be utilized with as wide a variety of communication facilities as possible.

The primitives that we utilize here consist of a simple bit string or "bucket of bits" communication facility. We assume that the active elements capable of communicating (hereafter referred to as processes) are able to send and receive bit strings (messages) of unlimited length and that the messages can be addressed to and from an unlimited supply of network addresses. We assume that the issues faced by typical end-to-end and lower level protocols (error detection and correction, flow and congestion control, routing and identification, etc.) are properly handled by the communication facility [Fle79, FlW78, Pos80, Wat79].

3. Defining the Distributed Resource Sharing Problem

The essence of the resource-sharing problem can be characterized by considering the three processes diagrammed in figure 4, a resource service and two resource-using processes. The problem can be divided into two parts: resource-access validation by the service process and resource-access passing between pairs of using processes.

For our discussion it will be convenient to have a single term to denote resource access. We will use the term capability [DeV66, Don76, Eng72, Fab74, Lan75, LaS76, Nee79, Wul74]. We say that a process has a capability to a resource if it has been authorized to access the resource.

This use of the term capability is a generalization of the way it is often used in the discussion of capability list (C-list) operating systems [Lan75, LaS76, Wul74]. In C-list systems capabilities do authorize resource access, but the term is generally restricted to refer to a specific form of capability implementation as some type of pointer that is protected by the kernel of a stand-alone OS. In such systems all capability passing and validation (in this restricted sense) is mediated by the system kernel. This centralized approach to access control unfortunately does not extend well into a distributed environment [Don76].

Our purpose here is to explore the types of protocols suitable for capability authorization and passing in a network operating system. Our protocols must allow serving processes to give out capabilities to potential users in such a way that:

a. The capabilities can be validated when used as authorization for service requests, and

b. The capabilities can be communicated or passed between any processes that can communicate data.

3.1 The Inalienable Right to Pass Capabilities

The requirement of capability passing (b above) often provokes some controversy. It is sometimes argued that allowing a process with an authorized capability to pass its authorization to another process is not always desirable. Indeed, a number of operating systems go to a great deal of effort to offer features that restrict capability passing.

Since the right to access any resource must originate from within the domain of the serving process, all capabilities must be passed at least once (namely from the service process to a using process) to be of any use at all. To handle this "first-pass" situation, some operating systems have used the expedient of a "pass-once" capability [Bas77]. Since a stand-alone OS can arrange to monitor capability pointer passing, it can mark "pass-once" capabilities as unpassable after they have once been passed from their service process to a user. In addition to being somewhat awkward, however, these attempts to restrict capability passing have the disadvantage that they can never really work. They can only work in the limited sense of the term capability discussed above (a pointer protected by a stand-alone OS), but not in

the more functional sense (that we have adopted) of granting the right to access a resource.

To see the difficulty of restricting capability passing we need only consider the processes A, B, and S pictured in figure 4. Suppose that A has a capability to a resource serviced by S. Also suppose that A can communicate with B (if not, then A clearly cannot pass capabilities or anything else to B; so no special capability passing restriction is necessary). If the mechanism used to pass direct access to a resource from A to B has been denied by a monitoring OS kernel, A can still give B the right to (indirectly) access the resource. A can simply have B send all its service requests to A for forwarding to S. A will also have to return the results of such requests to B.

In some cases this sort of indirect capability passing is a useful mechanism. For example in cases where A wishes to monitor B's accesses to the resource or where A wishes to be able to cut off B's access at a later time. In many other cases, however, the inability to pass direct access to a resource simply places an unnecessary additional communication burden on the passing process.

Since it is our intent to make the capability management protocols we consider as convenient as possible and since it is not possible to truely stop capability passing between processes able to communicate, we will only consider protocols that support the passing of direct resource access. Indirect capability passing (as above) is always available to any process that chooses to use it in lieu of direct capability communication.

4. Tools Available for Capability Validation

In choosing a capability management protocol it is convenient to center attention first on the validation mechanisms available. The capability passing mechanisms seem to follow naturally from the validation techniques chosen.

When a service process receives a request for resource access, it receives two pieces of information that can be used for capability validation:

i) the message data (the string of bits), and

ii) the address of the sender (remember we assume that our communication primitives validly identify the addresses of sending processes).

These two pieces of information naturally give rise to two basic approaches for capability validation.

4.1. Data Authorization

A service process using data authorization gives some secret information to processes authorized to access the resource. It hopes that other unauthorized processes will not be able to guess or surepticiously obtain the secret information.

4.1.1. Password Protected Capabilities

A simple form of data authorization is a pure password protocol. To create a password-protected capability, a service process creates a block of data containing the resource identification* and a secret password to authorize resource access. To validate an access request authorized by a password-protected capability, the service process need only compare the password in the received capability data block with the valid authorizing password. The authorizing password can either be choosen at random and stored with the resource or can be computed from the resource identification by a secret algorithm known only to the server (e.g., encrypted with a secret key).

4.1.2 Communicating Password-Protected Capabilities

Password-protected capabilities are the easiest type of all to communicate. To pass the right to access a password-protected capability from one process to another, it suffices to pass the capability data block (including its password) in a message.

4.1.3 The Data Theft Problem for Capabilities

The ease of communicating password-protected capabilities also gives rise to their most signifigant problem. A process with a password-protected capability in its domain (e.g. in its memory space) must be careful not to reveal the secret password. Although the domains of processes are generally protected from information theft, password-protected capabilities make some of a process's information particularly sensitive. For example, a resource such as a file or directory may remain in existence long after the process and its memory are no longer used.

The problem of information theft is especially serious in light of typical programming practices. It is quite common when debugging programs to output blocks of memory to examine them for program inconsistencies. If such memory output contains password-protected capabilities to sensitive resources, then it must be carefully protected (e.g. not taken to a consultant for analysis).

4.2 Address Authorization

A service process using address authorization utilizes the address of the requesting process (as supplied by the communication facility) in determining whether or not to allow a resource request.

---

* Resource identification is generally divided into a portion containing the address of the service-process and some service-process dependent additional information identifying a specific resource to the server. Since we are concerned here predominately with authorization and validation, we will not further discuss resource identification. Additional discussion of identification issues can be found in [FIW80, Wat80].

4.2.1 Access List Protected Capabilities

The conceptually simpliest form of address authorization is an access list protocol. To create an access-list-validated capability, a service process creates and maintains a list of addresses that are allowed to access the resource. Service processes for access-list-protected capabilities need only pass resource identification information to authorized using processes. This information is not used for authorization, however. To validate an access request authorized by an access-list-protected capability, the service process checks to see if the sender's address is in the access list for the identified resource.

4.2.2 Communicating Access-List-Protected Capabilities

Communicating access-list-protected capabilities is a good deal more difficult than communicating password-protected capabilities. The basic difficulty is that the service process must update its access list every time a capability is passed from one using process to another. To do this the service process must be notified that the capability transfer is taking place.

4.2.3 The Reflection Problem for Capabilities

At first glance it might appear that the passing of an access-list-protected capability could be accomplished in two steps:

S1. Request the service process to add the address of the process to receive the capability to the access list, and

S2. Send the resource identification to the receiving process.

Unfortunately, the above steps do not quite suffice for secure capability passing. The problem that exists is best illustrated by an example that is also useful in characterizing similar problems in other capability passing protocols.

The essence of the capability reflection problem is the situation that can arise if a process, D, can be duped into falsely believing that it received a capability from another process, B. If D can then be called upon to send (reflect) the capability back to B, B may receive an unauthorized capability.

To illustrate the capability reflection problem for the two-step access list protocol above, consider the directory service D depicted in figure 5. The function of this directory server is storage and retreival of capabilities. It allows processes to store capabilities in directories so that they may later be retrieved and used by any process with a capability to the directory. A directory server allows processes to store and share capabilities in a manner analagous to the way a file server allows storage and sharing of information.

We suppose for our example that process A has the only capability to an access-list-protected resource serviced by S. We also suppose that A has a capability to directory D_A and that B has a capability to D_B (but not to D_A). Any authorization protocol used for the

capabilities to the directories themselves will suffice. For example, the reader might consider them protected by passwords.

Now consider what must happen if A is to store its capability in D_A. Following steps S1 and S2 above, it first requests S to add D to the access list for the resource (1). A then sends the resource identification to D and requests that the capability be stored in D_A (2).

If at a later time some other process with access to D_A requests retreival of the stored capability, D must go through steps S1 and S2 to satisfy the request. That is it must have S add the retreiving process to the resource access list and then must return the resource identification to the retreiving process.

The capability reflection problem appears when we consider what happens if process B sends the resource identification (which A has not kept secret because it trusts the access list mechanism) to D and requests that its capability be stored in D_B (3). Of course B cannot succeed if it requests S to add D to the resource access list (since B itself is not on the list), but no matter. As we have seen, D is already on the list thanks to A. At this point the situation appears to D as if B were authorized to access the resource. All B need do is ask D to retreive the capability from its directory (4) and it will have duped D into placing it on the access list without proper authorization. We will see a similar reflection problem appear in conjuction with data theft when we consider public key encryption protection of capabilities.

4.2.4 Protecting from Capability Reflection

To supply an access-list-based capability management protocol that is safe from the reflection problem, we need to add something to the simple two-step protocol given above. One method of bolstering the two-step protocol requires all capabilities to be received from the resource service process. Using this protocol, a process sending a capability to another just notifies the receiver to expect the actual capability from the resource service process. The sender also requests the server to add the receiver to the resource access list AND to send the capability identification to the receiver on the sender's behalf. The fact that the process receiving the capability actually receives the resource identification from the service process along with the address of the sending process guarantees that the sender was authorized to access the resource.

This access-list protocol is somewhat awkward and requires a signifigant amount of message traffic. An additional defect is its requirement that the resource service process be available for capability transfers between any two processes at any time. The access-list protocol also requires servers to maintain access-lists which must be stored, checked for overflow, etc.

The advantage of the access-list protocol over password protection is its freedom from the data theft problem. The only information that can be stolen from a process's memory space concerning an access-list-protected capability is its resources' identifications. Since a resource identification is not sufficient authorization for resource access or capability passing, data theft is not a threat.

4.3 Capability Authorization Using Both Data and Address Authorization

There are a number of methods for combining data and address authorization in a capability management protocol. Two examples are now discussed.

4.3.1 Encrypted-Address-Protected Capabilities

This protocol is able to obtain the data theft protection of access lists without the need to manage the lists. In this scheme the resource service process grants a capability to a using process by encrypting the authorized user's address and the resource identification into a block of data that is sent to the authorized process (the portion of the data block containing the server's address is not encrypted). To validate resource access using this scheme the service process decrypts the data blocks sent in to authorize resource access and compares the decrypted address from the data block with the address of the sender returned by the communication facility. Only if the addresses match will the sender be authorized access to the resource identified in the decrypted data block.

4.3.2 Communicating Encrypted-Address Capabilities

To communicate an encrypted-address capability it must be passed to the receiver through the service process as with access-list-protected capabilities. With an encrypted-address protected capability, however, the server need not maintain an access list. Instead, it decrypts any capabilities to be passed, and, if the capability is valid, sends a capability to the receiver that is reencrypted with the receiver's address.

Encrypted-address-protected capabilities share with password-protected capabilities the hazard (however remote) that a capability could be guessed. As with access-list-protected capabilities, they require the cooperation of the service process for all capability transfers and are somewhat awkward for the receiver. In addition they require the overhead of the decryption and encryption operations when being passed.

The major advantage of encrypted-address capabilities is that they offer essentially the same resistence to the data theft problem as access-list-protected capabilities, but do not require the management of access lists by the service processes.

4.3.3 Public-Key-Encryption-Protected Capabilities

This last example authorization method provides the safety from data theft of access-list protection, but does not require access-list maintenance nor message exchanges with the service process for communication of capabilities.

This method depends for its operation on the existence of a practical public key encryption mechanism. We assume that every process A has a secret decryption algorithm which we will denote by $A_{\uparrow}$ ($_{\uparrow}$ = up

into the bright light of day). We also assume that ALL processes are able to encrypt data to be sent to any process A with a public encryption algorithm that we will denote by $A_{\Downarrow}$ ($_{\Downarrow}$ = down into the dark world of encryption). We therefore have that for any process A and data d, $A_{\Uparrow}A_{\Downarrow}d = d$. Finally, we assume that $A_{\Uparrow}$ and $A_{\Downarrow}$ commute, i.e. $A_{\Downarrow}A_{\Uparrow}d = A_{\Uparrow}A_{\Downarrow}d = d$. The reader is referred to [Lem79] for further discussion of such algorithms.

The basic idea of public-key-encryption protection is similar to that of encrypted-address protection. While a capability resides in the domain of a process, A, it is protected from theft by encryption. In this case instead of having A's address encrypted in the capability, a password-protected capability to the resourcec, $S_{\Uparrow}c$, encrypted with the public encryption algorithm of A, $A_{\Downarrow}S_{\Uparrow}c$. Because of the secrecy of $A_{\Uparrow}$, no process able to steal the encrypted capability will be able to use it. In order to avoid the problems associated with distributing the public keys [PoK79] the public key for the server is kept in clear text in the data block of every capability. Also, c must have enough redundancy so that a random data block will not be interpreted as a valid (random) capability.

4.3.4 Communicating Public Key Encrypted Capabilities

It might appear that for communicating a public key encrypted capability from A to B it would suffice for A to transform the stored $A_{\Downarrow}pc$ with $B_{\Downarrow}A_{\Uparrow}$. This would remove A's protection and reprotect it for B. Unfortunately, however, this simple form of capability communication is subject to a hazard of data theft and reflection. If a process could steal a capability from A's domain, it could reflect it off of A (send it to A and ask for it back as in the directory example given previously) and gain unauthorized access. Similarly, if a process could steal the capability from A after it had been readied for transfer to B, it could be reflected off of B to obtain unauthorized access.

These difficulties can be remedied by performing the following transformations for sending and receiving capabilities. For A sending a capability to B, it uses the transformation $B_{\Downarrow}A_{\Uparrow}A_{\Uparrow}$. The first $A_{\Uparrow}$ removes A's protection, the second $A_{\Uparrow}$ effectively signs the capability as coming from A, and the final $B_{\Downarrow}$ protects it so that it cannot be stolen from B's domain when it arrives.

For B receiving a capability from A, it uses the transformation $B_{\Downarrow}A_{\Downarrow}B_{\Uparrow}$. The first $B_{\Uparrow}$ undoes the protection that A put on for B, the middle $A_{\Downarrow}$ unsigns the capability, and the final $B_{\Downarrow}$ protects the capability for residence in B's domain. Figure 6 follows the transformations performed as a capability for a resource identified by c passes from a server S to A, then to B, and finally back to S for authorization.

An interesting property of this protocol is the fact that a process with a capability can safely "look" at the clear text identification of the resource. For example, in figure 6 process A can apply $S_{\Downarrow}A_{\Uparrow}$ to the $A_{\Downarrow}S_{\Uparrow}c$ that it holds in memory to yield c. There is no danger from having the clear text c stolen (at least in terms of protecting the resource-access of c) because no process will be able to produce the $S_{\Uparrow}c$ that is required for validation by S. Another interesting and convenient fact is that while the capability is in the servers domain

its normal storage form is clear text. The server's "look" transformation is the identity transformation.

An important requirement of this protocol is that its transformations (receiving, looking, and sending) be indivisible. That is, intermediate results must not appear in the memory of processes where they might be stolen. For example, if the transformation $B_{\Downarrow}A_{\Uparrow}A_{\Uparrow}(A_{\Downarrow}S_{\Uparrow}c)$ were to yield the intermediate result $S_{\Uparrow}c$ in memory after the first $A_{\Uparrow}$, the integrity of the mechanism would be breached. $S_{\Uparrow}c$ is essentially a password-protected capability in that any process able to steal it, X, can transform it into $X_{\Downarrow}S_{\Uparrow}c$ and then use it or pass it.

The protection of the private encryption key and the intermediate results can be achived in several ways. For example in a multiprocessing OS component the transformations can be performed by the OS kernel in responce to a virtual user instruction (only the kernel knows the process's private decryption key). In a smaller single domain system (e.g. a microprocessor system) it might prove effective to have the transformations performed in a hardware device that alone knows the system's private decryption key.

It has been pointed out elsewhere that in many situations the problem of distributing public encryption keys is as severe as that of distributing secret keys [PoK79]. The difficulty comes from trying to match a public key with some previously identified entity. For example, is this public key really that of the person with whom I intent to correspond? Fortunately this difficulty is not present in the capability protection mechanism discussed here since the public keys are distributed in the capability data blocks. It is true that a server (or anyone else) could endanger a trusting user by passing out a capability with an incorrect public key or other invalid identification information. This is simply an instance of the well known Trojan Horse problem, however, and is necessarily present in the best of capability management mechanisms.

This public-key-validation protocol has the strength from data theft of the access-list and encrypted-address schemes, but requires no access-list management or extraneous message passing. Its major weakness is the overhead involved in the transformations required. It depends, of course, on the existence of a suitable public key encryption algorithm.

It should be mentioned here that there are capability protection protocols that protect from data theft without message traffic but do not depend on public key encryption. The only such protocols known to the authors [Nes80], however, involve some additional complications for key distribution and and management that are not relevant here.

7. A Capability Passing Structure

We have explored four examples of capability passing protocols. Each seems to have strengths and weaknesses. From the viewpoint of a process wishing to communicate capabilities this is a rather negative result. It appears that there is little likelihood of finding a best capability passing protocol that can be used as a standard. There is a great deal of commonality in these example protocols, however. By

exploiting this commonality it may be possible to at least define a common capability passing structure.

All of our example capability passing protocols can be divided into three analagous parts: the communication proper, the sending transformation, and the receiving transformation. The communication proper is simply a matter of communicating the capability data block including the server's network address and the server dependent information. The sending and receiving transformations break down as pictured in table 1.

Table 1

Sending and Receiving Transformations for Example Capability Passing Protocols

	Example	Sending	Receiving
a.	Password	None	None
b.	Access List	Request Server to add to list and forward ident.	Receive identification from server.
c.	Encrypted Address	Request server to reencrypt and forward.	Receive reencrypted capability from server.
d.	Public Key A ← B	$B_{\downarrow}A_{\uparrow}A_{\uparrow}$	$B_{\downarrow}A_{\downarrow}B_{\uparrow}$

From the table it is apparent that examples b and c are identical from the perspective of the communicating processes. In both cases the sender also sends the capability to the server and in both cases the receiver receives the capability confirmation from the server.

To integrate these example protocols into a common structure we can supply each capability passing process with library routines for the sending and receiving transformations that can handle the three distinct forms. The routine "SendTransform" is passed the stored capability data block and the address to which the capability is to be sent. It returns a capability data block suitable for direct transfer. The capability data block must always keep a description of the form of protection that it uses in clear text. This protection form description allows the transformation routines to determine which transformation to perform.

The routine "ReceiveTransform" is passed the received capability data block and the address that the block was received from. It returns the transformed capability data block in a form suitable for storing in the domain of the receiving process.

This common structure highlights the practical difficulties with the access list and encrypted-address protocols. In general the receiving transformation for this capability passing form must wait for a message from the server before being able to store the transformed capability.

8. Integration of Network Directories

It is a little difficult to fit the capability management protocols into perspective without an example of their use. Accordingly we give the example of the directory service discussed briefly earlier. This example illustrates the the type of network-wide name space needed for humans to share capabilities effectively.

The basic idea of the directory service is to store capabilities by name (figure 5). The essential operations on a directory resource are store, retrieve, delete, and list:

Table 2

Primitive Directory Operations

Operation	Sends directory capability and opcode and	Directory server returns
a. Store	the capability to store and the name to store it under.	OK or error (e.g. name conflict).
b. Retreive	the name to retreive	the capability retreived
c. Delete	the name to delete	OK or error
d. List	-nothing else-	a list of stored capability names

A capability to a directory is one type of capability that can be stored in a directory. Directories therefore naturally form a general directed graph (figure 7 and 9). Furthermore, since a capability to a directory serviced by one server can be stored in a directory serviced by another, the directory structures are trivially integrated (figure 7). The ability to integrate directories in this way is a consequence of the fact that the directory servers share a common capability-passing protocol.

8.1 Human Use of Directories

A natural way to introduce human beings into a resource sharing structure managed by directories is to give each user a private directory in which to store capabilities. Since many of the resources stored in this private directory will themselves be directories, the private directory becomes the starting point or root from which the other resources belonging to a user can be reached. Each user can be given a password-protected capability to this root directory to memorize and later regurgitate when "logging in."

If each user is also given some type of account capability to authorize creation of new resources* then they can get started building a resource structure for their private use. Directories can be created for storing related resources and saved in the root directory for future reference. Files, processes, or other resources can be stored in this created directory structure.

8.2 Directory Manipulation Utilities

Typically users manipulate their directory structures with a general purpose utility that performs operations such as:

a. Duplicate $Path_1$ to $Path_2$

b. Delete $Path_1$

c. Create Directory $Path_1$

d. Destroy Directory $Path_1$

Here the Paths are lists of names, $Name_1$,$Name_2$, ..., $Name_{PathSize}$. For all the paths the utility starts with the user's root directory and scans down successive names as noted in figure 8. The resulting directory, Dir, and the last name, Name, are used by the operations as:

a. Duplicate - Retrieve $Name_1$ from Dir_1 and store it in Dir_2 as $Name_2$.

b. Delete - $Name_1$ from Dir_1.

c. Create - a directory serviced by a convenient directory server and store its capability in Dir_1 as $Name_1$.

d. Destroy - Retrieve the capability $Name_1$ from Dir_1. Assume that it is a dirctory capability and have the server destroy it. Also delete the now useless capability $Name_1$ from dir_1.

---

* Discussion of network accounting mechanisms is beyond the scope of the present paper. Such mechanisms are being actively investigated at LLL. We expect to report the results of these investigations at a later time.

These interactive services are quite useful for manipulating a directory structure. Unfortunately, however, if the directory structures of two users are disjoint (neither contains a directory of the other), these primitives cannot be used by the users to share resources. Users can begin sharing with each other by communicating capabilities process to process, but this is somewhat awkward and outside the scope of the services offered by the typical directory manipulation utility.

8.3 Bootstrapping Resource Sharing with Directories, Give and Take Directories

Once the directories of two users are linked, sharing of resources may be possible using a directory manipulation utility as above. If the user directory structures are linked appropriately, the initial linkage can be used to bootstrap further more flexible sharing.

At LLL the Elephant Storage System [FIF75] (which never forgets user directories) has for many years supported a general directed graph directory structure to facilitate sharing of user resources. The bootstrapping linkage used in this system is simple but powerful. It provides two basic facilities:

i) Any user can give a capability to any other user, and

ii) Any user can make a capability available to all other users.

To supply i) each user is given a directory that others can use to give him capabilities. A capability to store into this "Give" directory is placed in a public directory under the user's name. A capability to list and retreive capabilities from the public directory of "Give" directories is also given to every user. By using the duplicate operation described above a user can give a capability to any other user if they are linked in this way.

To supply ii) each user is given a directory that he can put capabilities into to allow others to take them. A capability to list and retrieve from this "Take" dirctory is placed in a public directory under the user's name. A capability to list and retrieve capabilities from the public directory of "Take" directories is also given to every user.

Starting from this simple bootstrap linkage, users can apply the directory operations described above to construct any type of shared directory linkage desired. This allows humans to conveniently share resources by using a network-wide naming structure built on top of the capability passing protocol.

9. The Current State of the Protocols at LLL

The network systems division at LLL is currently in the process of upgading the communication protocols for the Octopus Network [FIW80] to support a tightly coupled network network operating system [Don79]. A multiprocessing component operating system is being implemented to support these new protocols. The emulation of this component system that is currently running uses only password-protected capabilities.

We expect to continue running the early versions of the network operating system using only password-protected capabilities (because of their simplicity) until a need for protection from the data theft problem arises. The current servers that distribute password-protected capabilities encrypt the resource identification into the capability data block using a software implementation of the National Bureau of Standards (NBS) Data Encryption Standard (DES) [NBS77].

For many years now the Elephant Storage System has provided a centralized directed graph directory structure for sharing user files and directories at LLL. When we shift our network protocols over to the capability passing structure described above we also expect to upgrade the central directory structure to store these generalized capabilities. As noted before this will automatically integrate it with the local directory services on the component "worker" systems such as the Cray-1. This integration will give both humans and processes at LLL the ability to share the Octopus network resources quite flexibly.

10. Conclusions

Many of todays mature resource sharing computer networks require an integrated network operating system to make effective use of their facilities. The operating system's task of controling access to its shared resources is greatly simplified if it can control resource sharing with protocols that are independent of the semantics of the many resources available in such a network. We have discussed some of the issues involved in the design of such protocols and have explored the strengths and weaknesses of a number of examples.

Some of the resource-access management mechanisms we have discussed are not new. Certainly the use of passwords to protect login access to time sharing systems must by now be considered an ancient tradition in the lore of computer science. Schemes similar to the access-list protection protocol can be found in the venerable ARPA network initial connection and file transfer protocols [ARP78] and in many more modern systems. However, the resource independent sharing protocols developed here go far beyond the protocols currently popular in distributed systems. As we illustrated with the integrated network directory example, there are many distributed system problems now considered difficult that become trivial when considered in the context of a common resource-access management protocol.

The main thrust of this paper has been the suggestion that access control protocols can and should be studied and implemented apart from the semantics of specific resources. The discussion has been carried out using the somewhat methaphorical terminology of communicating or passing capabilities that has proven effective in use at LLL and elsewhere.

It has become apparent to us at LLL that the success of our network operating system implementation will depend to a large extent on the success of our distributed access control protocol(s). We expect such protocols to become a more important part of future efforts in distributed system design.

11. Acknowledgements

One historical line of development for the capability sharing protocols we have discussed can be traced from [DeV66] through experimental C-list operating systems at MIT and at LLL. The LLL system, RATS [Lan75], was implemented by Charles Landau with the assistance of one of the authors (Donnelley). Another historical line of development can be traced to some early directory ideas from Multics [Org72] that were built into the Elephant Storage System at LLL [FlF75] by Garrett Boer with the assistance of one of the authors (Fletcher).

The authors gratefully acknowledge the contributions from collaboration with their confederates in network operating system design at LLL: Dick Watson, Lance Sloan, Bob Crallee, Pete DuBois, Chuck Athey, Donna Mecozzi, and Jim Minton.

12. References

[ARP78] *ARPAnet Protocol Handbook*, Network Information Center, SRI International, Menlo Park, Calif. 1978.

[BaS77] Baskett, F., et. al., "Task Communication in Demos," *Proc. of the Sixth Symposium on Operating System Principles*, *ACM*, Purdue University, 1977, pp. 23-31.

[Bob72] Bobrow, D. G., "Tenex, A Paged Time Sharing System for the PDP-10," in *Comm. ACM Vol. 15*, No. 3 (March 1972), pp. 135-143.

[Chr77] Christensen, G. B., "Hyperchannel Data Trunk Contention," in *Proc. 2nd Conf. on Local Computer Networks*, 1977, University of Minnesota, Minneapolis.

[DeV66] Dennis J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Comm. ACM Vol. 9*, No. 3 (March 1966), pp. 143-155.

[Don76] Donnelley, J. E., "A Distributed Capability Computing System," *Proc. of the Third International Conference on Computer Communication*, *ICCC*, Toronto, Ontario (Aug. 1977), pp. 432-440.

[Don79] Donnelley, J. E., "Components of a Network Operating System," in *Proc. 4th Conf. on Local Computer Networks*, IEEE 79 CH 1446-4C, (Oct. 1979), pp. 1-12 (to appear in *Computer Networks* in 1980).

[DoY76] Donnelley, J. E. and J. Yeh, "Interaction Between Protocol Levels in a CSMA Broadcast Network," in *Computer Networks* Vol. 3, North Holland, 1979, pp. 9-23.

[Eng72] England, D. M., "Architectural features of system 250," in *Infotech state of the art report 14: operating systems*, 1972, Infotech International Ltd., Maidenhead, Berkshire, England, pp. 395-428.

[Fab74] Fabry, R. S., "Capability Based Addressing," *Comm. ACM* Vol. 17, No. 7 (July 1974), pp. 403-412.

[Fle73] Fletcher, J. G., "The Octopus Computer Network," in *Datamation* Vol. 19, No. 4 (April 1973), pp. 58-63.

[Fle79] Fletcher, J. G., "Serial Link Protocol Design: A Critique of the X.25 Standard, Level 2," *Lawrence Livermore Laboratory Report UCRL-83604*, October 1979.

[Fle80] Fletcher, J. G. and R. W. Watson, "Service Support in a Network Operating System," in *COMPCON '80 Conf.*, *IEEE*, San Francisco, Calif., March 1980.

[FlF75] Fletcher, J. G., et. al., "Computer Storage Structure and utilization at a Large Scientific Laboratory," in *Proc. IEEE* Vol. 63, No. 8 (Aug. 1975), pp. 1104-1113.

[FlW78] Fletcher, J. G. and R. W. Watson, "Mechanisms for a Reliable Timer-Based Protocol," *Computer Networks* No. 4/5 (Sept./Oct. 1978), pp. 271-290. Also in *Proc. Computer Network Protocols Symposium, Liege, Belgium* (Feb. 1978), p. C5-1/C5-17.

[FlW80] Fletcher J. G. and R. W. Watson, "Service Support in a Network Operating System," in *COMPCON '80 Conf.*, *IEEE*, San Francisco, Calif., March 1980.

[Lan75] Landau, C. R., *The RATS Operating Systtem*, Lawrence Livermore Laboratory Report UCRL-77378 (1975).

[LaS76] Lampson, B. W. and H. Sturges, "Reflections on an Operating System Design," *Comm. ACM Vol. 19*, No. 5 (May 1976), pp. 251-265.

[Lem79] Lempel, A., "Cryptology in Transition," *Computing Surveys, ACM* Vol. 11, No. 4 (Dec. 1979), pp. 285-303.

[NBS77] National Bureau of Standards, *Federal information processing standards*, Publ. 46.

[Nee79] Needham, R. M., "Adding Capabilities Access to Conventional File Services," *ACM Operating Systems Review*, Vol. 13, No. 1 (Jan. 1979), pp. 3-4.

[Nes80] Nessett, E., "A Preliminary Reveiw of Three Secure Capability Passing Mechanisms," private communication.

[Org72] Organick, E. I., *The Multics System: An Examination of Its Structure*, Mit Press, Cambridge, Mass. (1972).

[PoK79] Popek, J. G. and C. S. Kline, "Encryption and Secure Computer Networks," *Computer Surveys, ACM Vol. 11*, No. 4 (Dec. 1979), pp. 331-356.

[Pos80] Postel, J. B., "DoD Standard Internet and Transmission Control Protocol Specification," IEN 128, 129 (Jan. 1980), Available through the Defense Advanced Research Project Agency, IPTO, Arlington, Va.

[RiT74] Ritchie, D. M. and K. Thompson, "The Unix Time Sharing System," in *Comm. ACM Vol. 17*, No. 7 (July 1974), pp. 365-375.

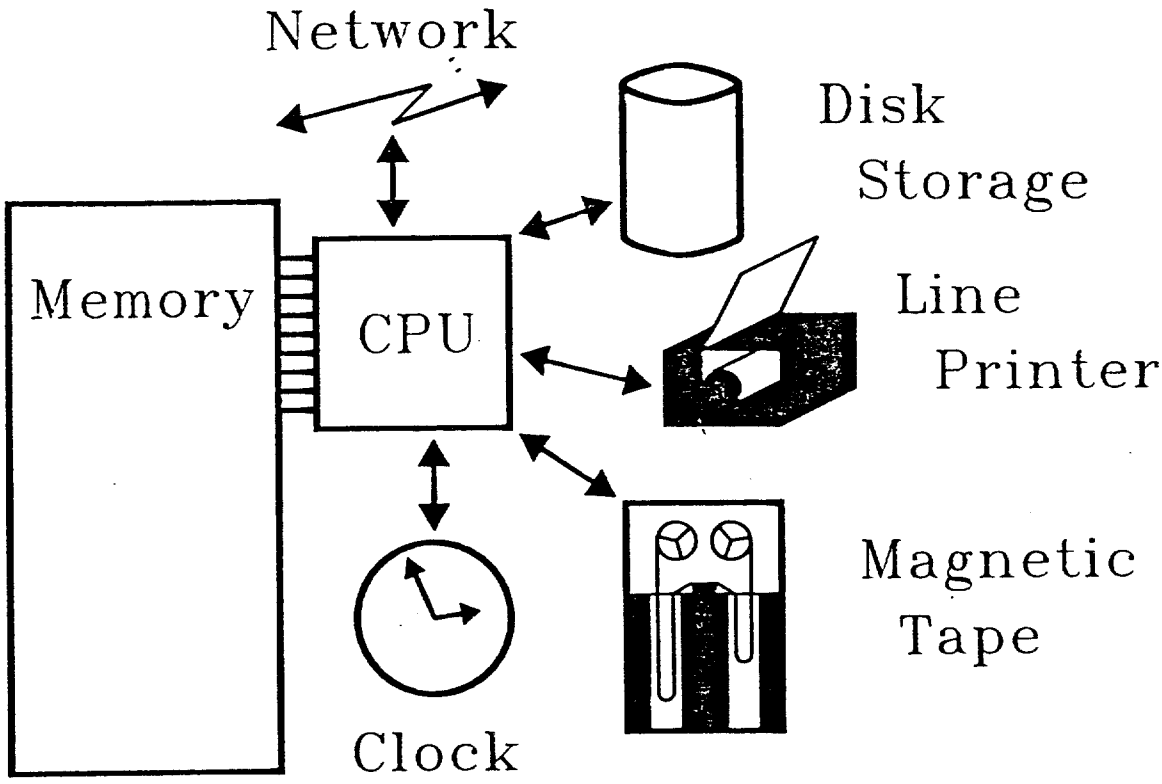
[Wat78] Watson, R. W., "The LLL Octopus Network: Some Lessons and Future Directions," in *Proc. Third USA-Japan Computer Conference*, San Francisco, Calif. (Oct. 1978), pp. 12-21.

[Wat79] Watson, R. W., *Delta-t Protocol Specifications*, Lawrence Livermore Laboratory Report UCRL-52881, Nov. 1979.

[Wat80] Watson, R. W., "Naming in Distributed Systems," to appear in *Distributed Systems, an Advanced Course*, Berlin/Heidelberg/New York: Springer-Verlag, 1980.

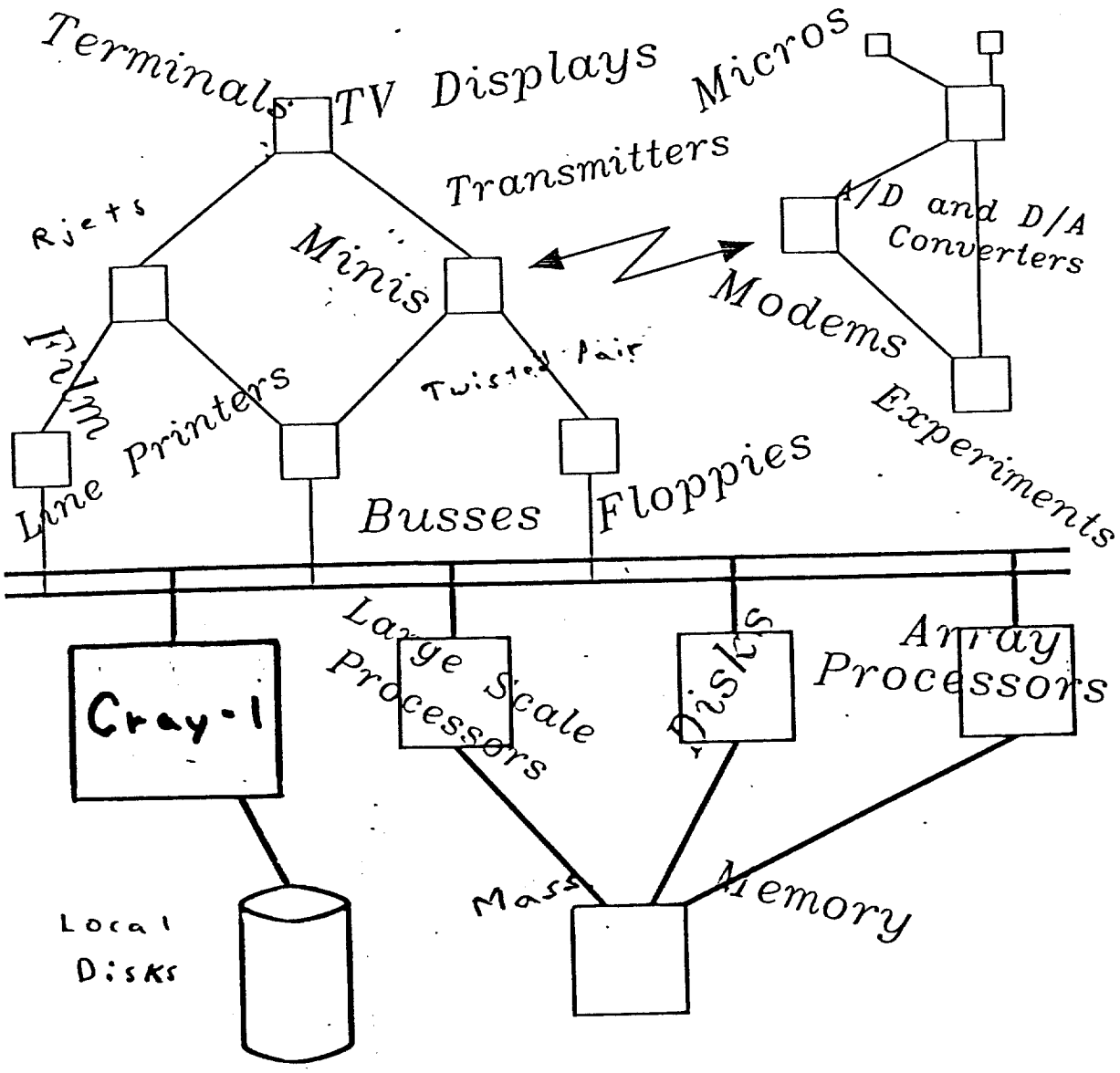
[Wul74] Wulf, W. A. et. al., "Hydra: the Kernel of a Multiprocessor System," in *Comm. ACM Vol. 17*, No. 6 (June 1974), pp. 337-345.

Figure 1



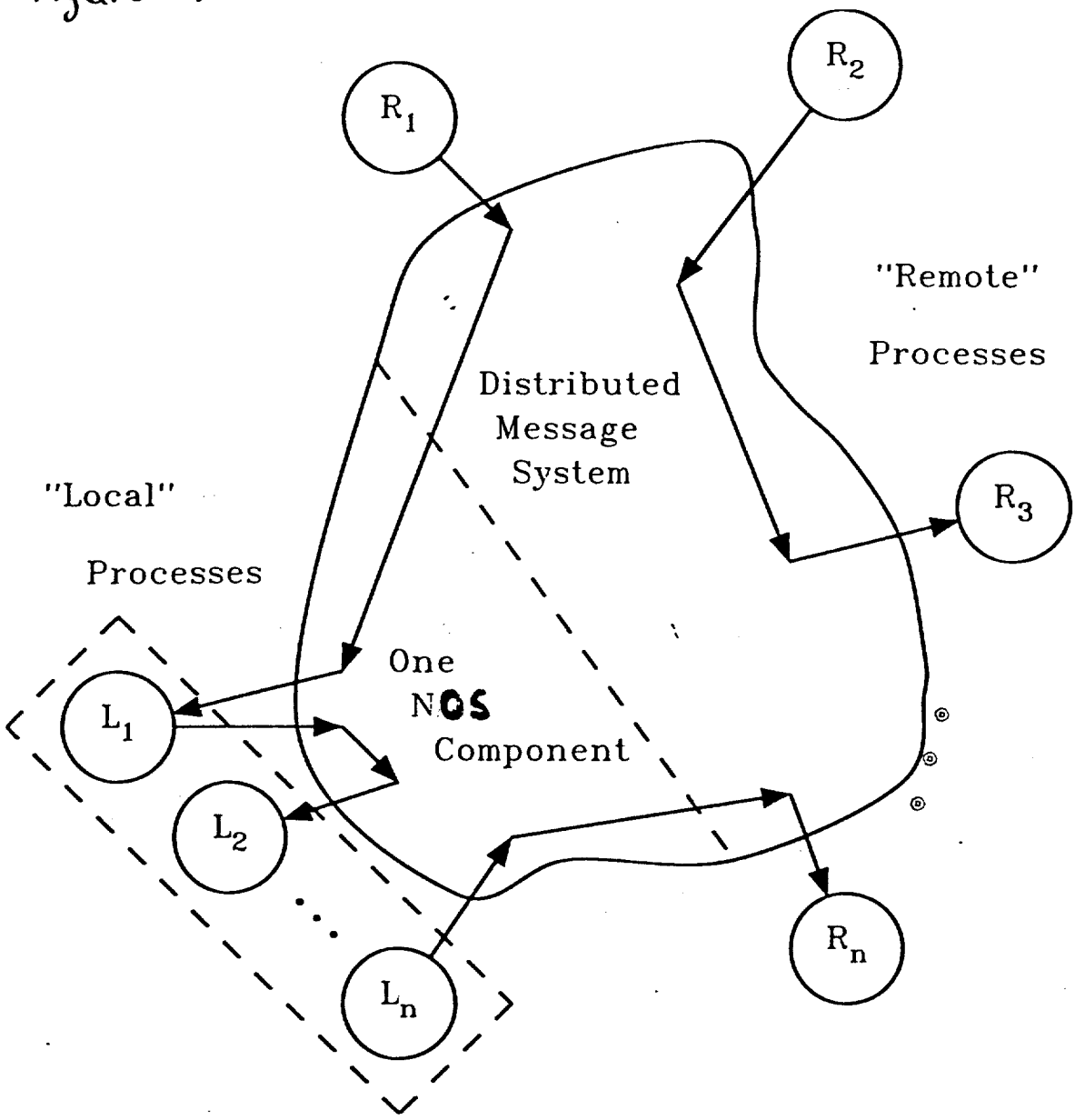
Stand-alone Computer System Multiplexing
Directly Attached Peripherals

Figure 2



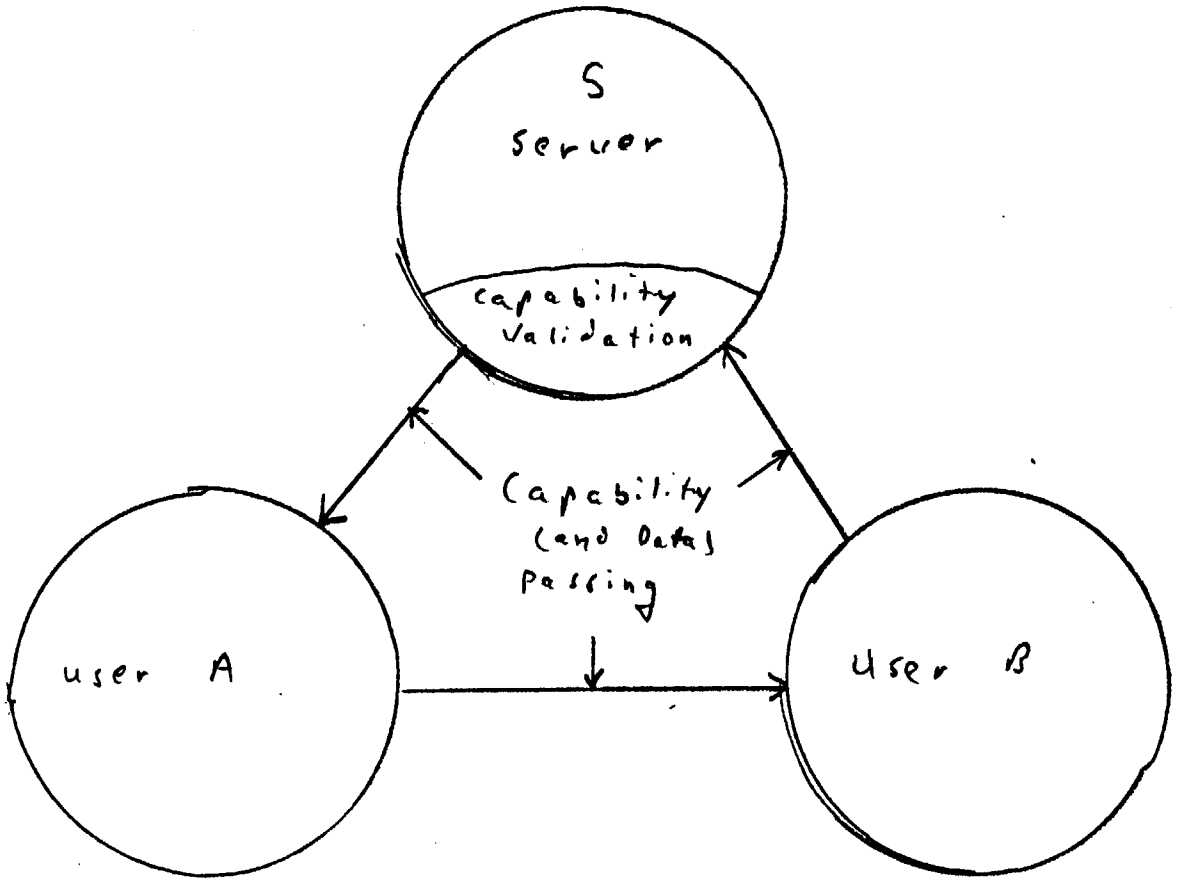
An Example Distributed Computing Facility

Figure 3



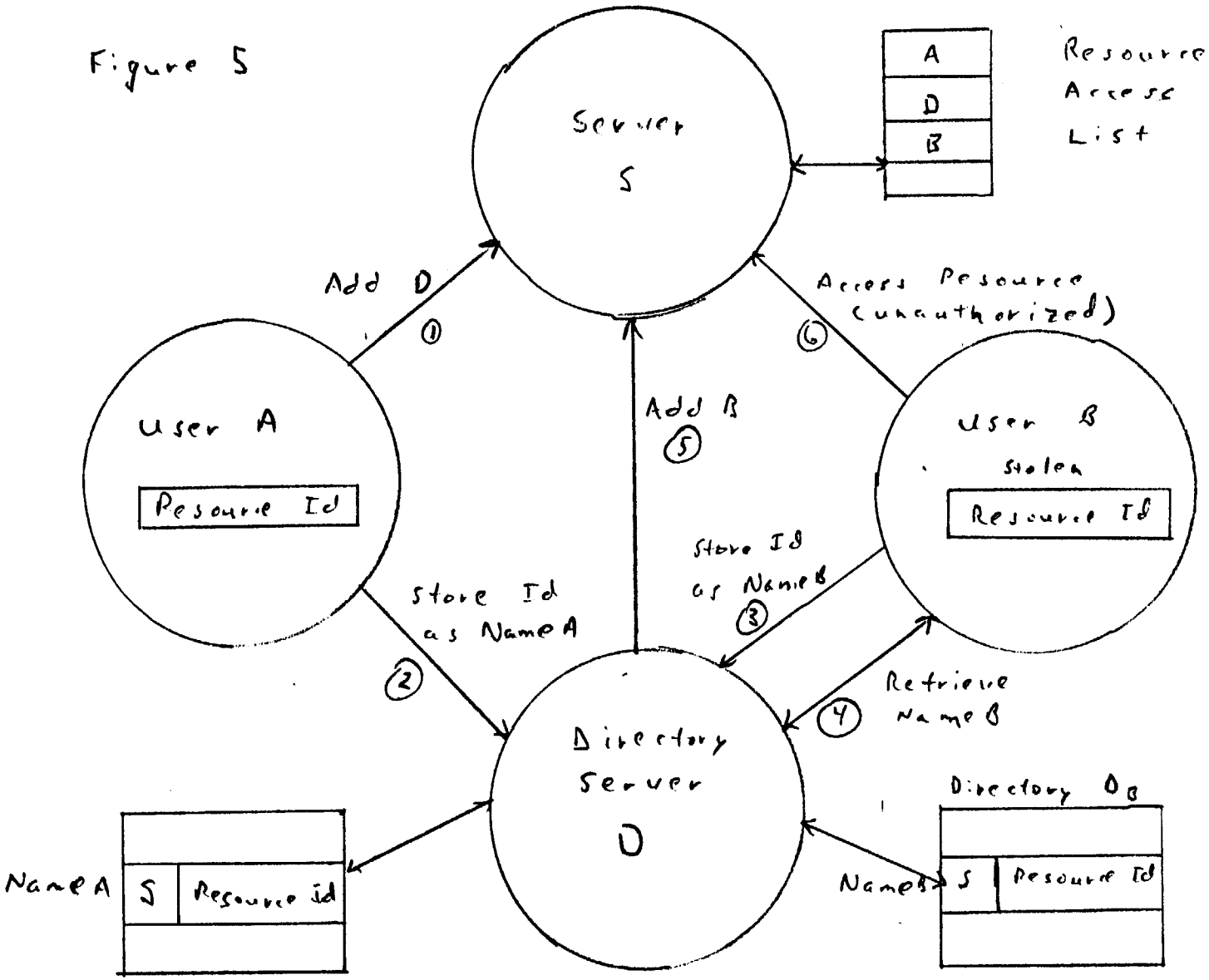
Sharing Resources over a Communication Network

Figure 4



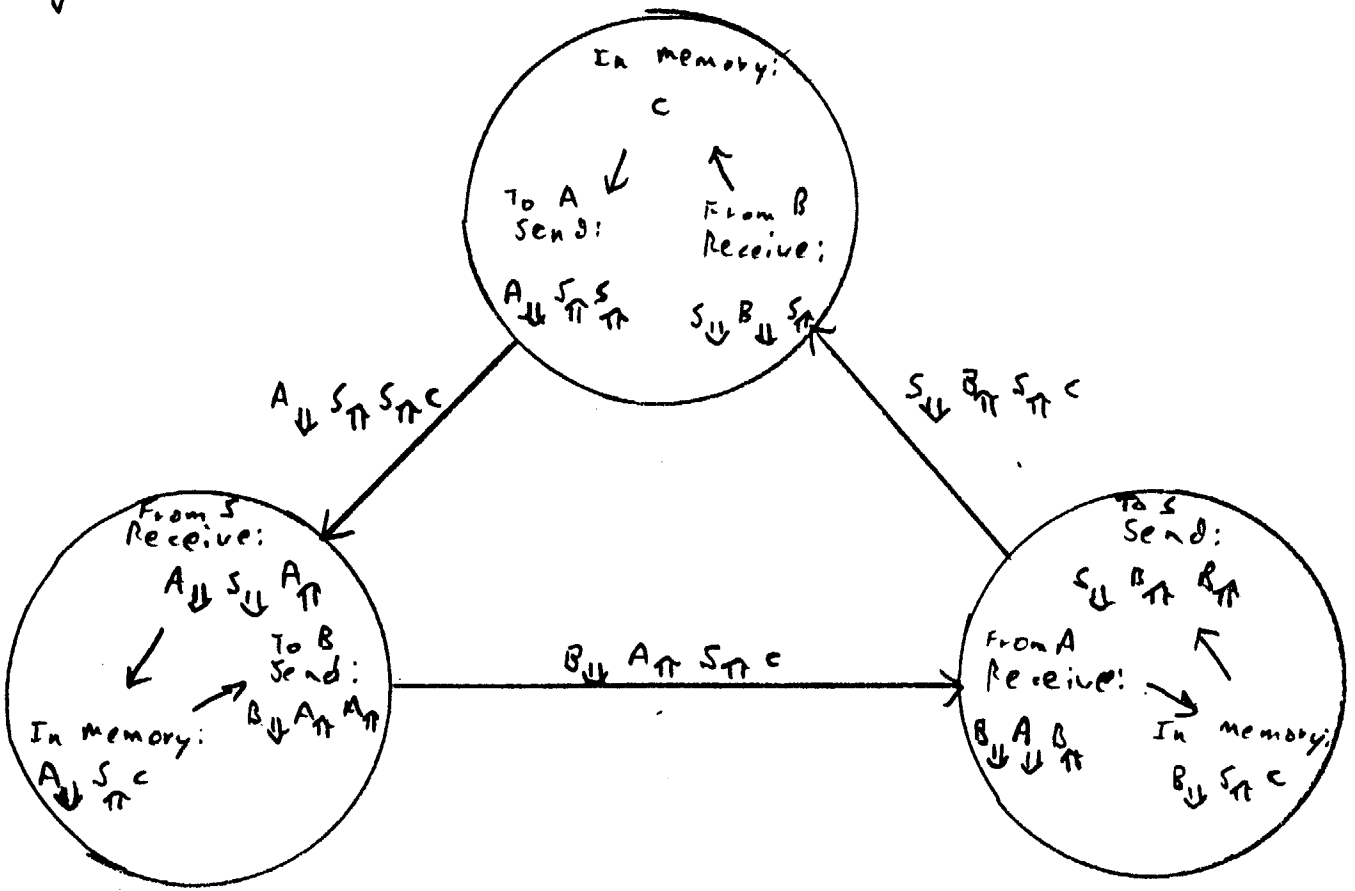
The Distributed Resource Sharing Problem

Figure 5



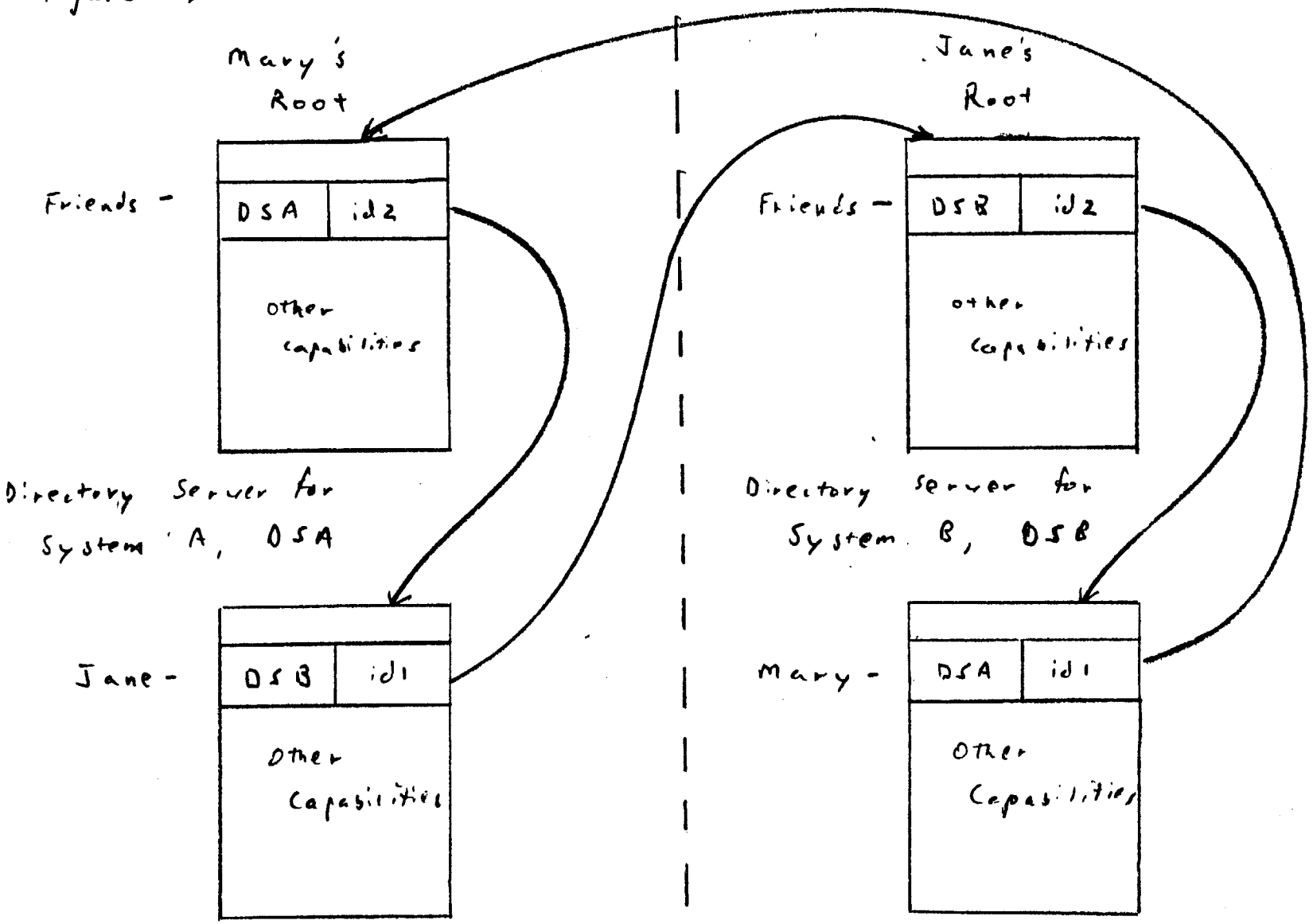
Capability Reflection Problem Example
The Directory Service

Figure 6



Passing Public-key-Protected Capabilities

Figure 7



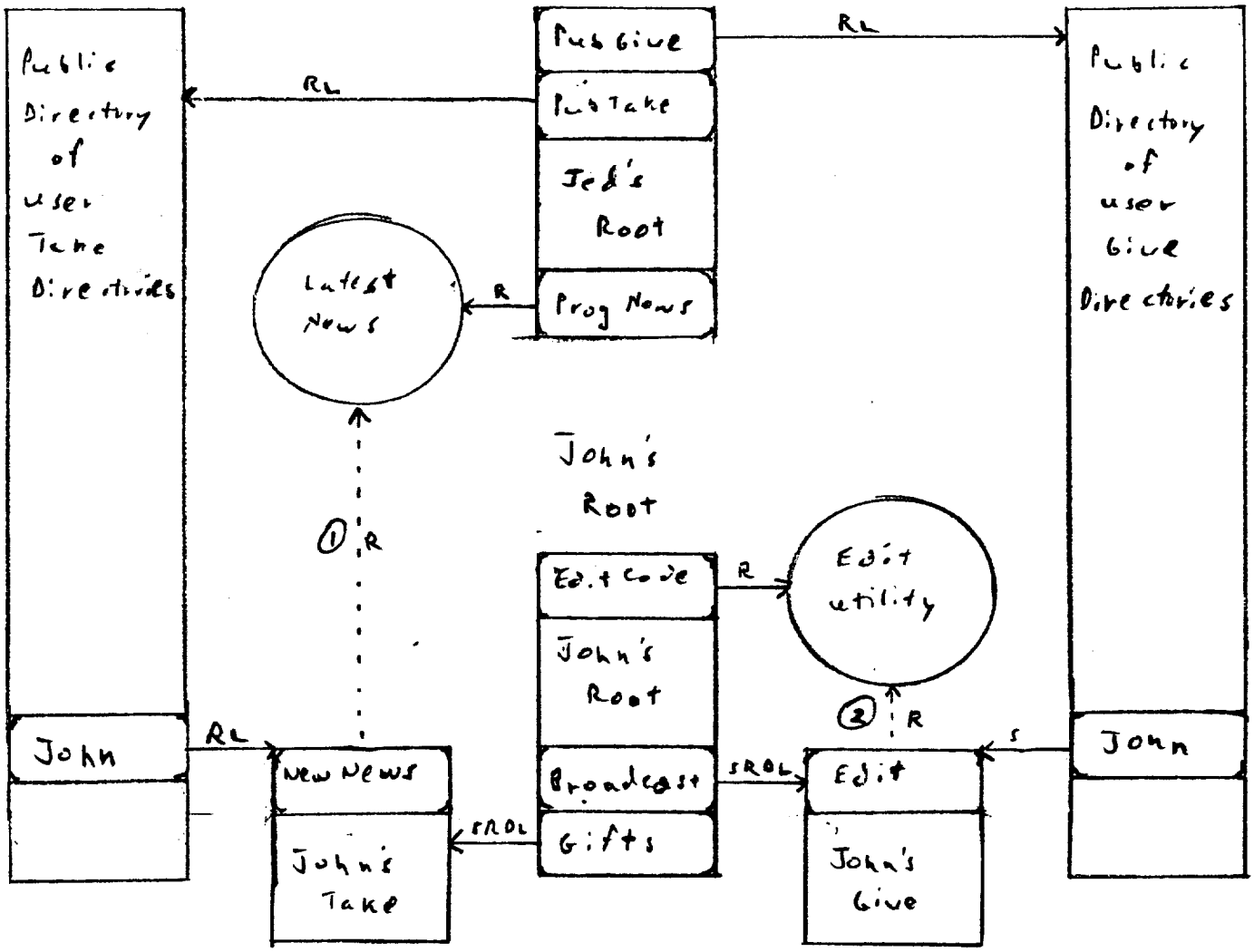
Integrated Network Directories, true Friends:
"Everything that's mine is yours."

Figure 8

```
Dir := UserRoot;
Name := Path[0];

For ScanIndex := 1  Step 1 Until PathSize - 1
Do
   Begin
      Dir := Retrieve (Dir, Name);
      Name := Path[ScanIndex];
   End;
```

Simplified Scan Loop for Paths



The Give and Take Directory Bootstrap

Figure 9 illustrates the following give and take examples:

Jed: "Why doesn't Edit use the new error code standard?"

John: "The changeover hasn't been announced yet."

Jed: "You must not have seen the latest programmer's news."

Duplicate ProgNews to PubTake,John,NewNews

"Look at the 'NewNews' that I just gave you."

John: *Display Gifts,NewNews*

"I see. I have a new version of Edit here, wait."

Duplicate EditCode to Broadcast,Edit

"Try it now."

Jed: *Run PublicTake,John,Edit*

<Test>

"That seems to do it. Thanks."